

## QUALITY WEEK - 1993

*Sixth International Software Quality Week (QW-93)*

**25-28 May 1993**

*Sheraton Palace Hotel • Market & New Montgomery • San Francisco, California*

### SPEAKERS & TITLES

The following lists confirmed speakers scheduled for **Quality Week 1993**, as of the current date. Included are regular papers (approx. 45 minutes), special 90 minute presentations, Invited Keynote Addresses, and 1/2-day Tutorials. The list is arranged in alphabetic order by speaker name.

Papers are classified according to their assigned track: (T) indicates the technical track; (M) indicates the management track; (A) indicates the application track. In addition, (Tutorial) indicates a 1/2-day Tutorial and (Keynote) indicates Invited Keynote Addresses.

O O O

**Mr. John Ambrose & Maily Pham** (*Sierra Semiconductor, San Jose, CA*): "Automated Firmware Testing Using Simple Test Scripts".

**Dr. Boris Beizer** (*ANALYSIS, Huntingdon Valley, PA*): "Overview of Software Test Techniques" *Tutorial*.

**Dr. Boris Beizer** (*ANALYSIS, Huntingdon Valley, PA*): "Two Retrospectives From the Year 2001" *Keynote*.

**Mr. Bill Bently** (*Miles Inc., Elkhart, IN*): "Information Flow Analysis"

**Mr. Mark Bergman** (*Tandem Computer*): "Automated Testing vs. Debugging" *Mini Tutorial*.

**Ms. Antonia Bertolino** (*CNR/IFI, Pisa, ITALY*): "Path Analysis and Experimentation on Path Coverage via Minimum Test Covers".

**Prof. Vern J. Crandall** (*Sun Microsystems, Mountain View, CA*): "Using Executive Modules as Test Drivers (tentative)".

**Mr. Bill Curtis** (*Carnegie Mellon University, Pittsburgh, PA*): "Maintaining the Software Process Movement" *Tutorial*.

Mr. Bill Curtis (*Carnegie Mellon University, Pittsburgh, PA*): "Superior Software Organization" *Keynote*.

Mr. Gregory T. Daich (*Software Technology Support Center, UT*): "Technology Infusion into the Software Test Process".

Mr. Peter H. Jettler (*Carnegie Mellon University, Pittsburgh, PA*): "Software Process Development and Placement: Concepts and Definition".

Mr. Mark A. Jewster (*Racal-Redac Systems Ltd., Gloucestershire GL20 8HP*): "100% Automated Testing: A Case History".

Ms. Francoise Fichoux-Vapne (*Electricite de France, 91004 Evry CEDEX*): "Software Capabilities Assessment: New Perspectives in Europe".

Mr. Tom Gibb O: "Advanced Software Inspections" *Tutorial*.

Mr. Tom Gibb O: "Practical Software Metrics for Software Process and Product Quality" *Mini-Tutorial*.

Dr. Dorothy R. Graham (*Grove Consultants, Cheshire*): "Where is a ST Heading? Directions and Trends for Testing Tools" *Keynote*.

Major Christine W. Maapala (*USAF (JNIDS), Washington, D.C.*): "Verification and Validation in an Interactive Software Development Environment".

Prof. Richard Hamlet & Dr. Jeffrey M. Voas (*Portland State University, Portland, OR*): "Foundations of Program Testing: Can Tested Software be Trusted?" *Mini-Tutorial* {Thursday or Friday}.

Prof. Mary Jean Harrold (*Clemson University, Clemson, SC*): (Subject is "object-oriented"; title being sent).

Dr. Herb Hecht (*SoMA, Inc., Beverly Hills, CA*): "Race Conditions - An Important Cause of Failures".

Mr. Jonathan Hops (*Jet Propulsion Laboratories, Pasadena, CA*): "Formal Functional Test Design: Bridging the Gap Between Test Requirements and Test Specifications".

Mr. Dean O. Hashizaki (*Microsoft Corporation, Redmond, WA*): "Risk Based Testing Application Testing".

Mr. William Howden (*University of California, San Diego, San Diego, CA*): "Scientific Foundations for Practical Software Testing and Analysis" *Keynote*.

Mr. Gilbert A. Iacono (*Jimbach International, Pittsburgh, PA*): "Automating Test Database Management for Maximum Productivity".

Ms. Karen S. King (*Cadence Systems Corporation, Beaverton OR*): "Implementing Software Process Improvement".

Jonathan M. Hops

Jet Propulsion Laboratory  
California Institute of Technology

**Abstract:** This article discusses the application of the Category-Partition Method [5] to the test design phase. The method provides a formal framework for reducing the total number of possible test cases to a minimum logical subset for effective testing. An automatic tool and a formal language have been developed to implement the method and produce the specification of test cases.

**Keywords:** software testing, test designs, software life cycle, testing lifecycle, formal test representation, test specifications, category-partition method, test representation language, TRL.

## 1. Introduction

The focus of this paper is how the category-partition method, a method for specifying functional tests [5], can be applied to the test design phase of the testing lifecycle. Before describing the method itself, we need to clearly define what the requirements are for the test design phase. This discussion can be found below, in Section 2.

Section 3 centers on some methods often used in the test design phase. The category-partition method is described in detail. Included in this section is the background of the method, a step by step description of how to implement it, and a demonstration of the method applied to a simple example.

The subsequent section, Section 4, introduces the Test Representation Language (TRL). TRL is a formal language for specifying test designs that have been created with the category-partition method and a computer tool for automatically generating test cases from the formal specification. The example from Section 3 is presented, using the TRL format.

The conclusions in Section 5 provide some insight into the results that have been achieved, and some suggestions for further study and data collection that may be necessary to assess the contribution of the TRL tool developed and the category-partition method used.

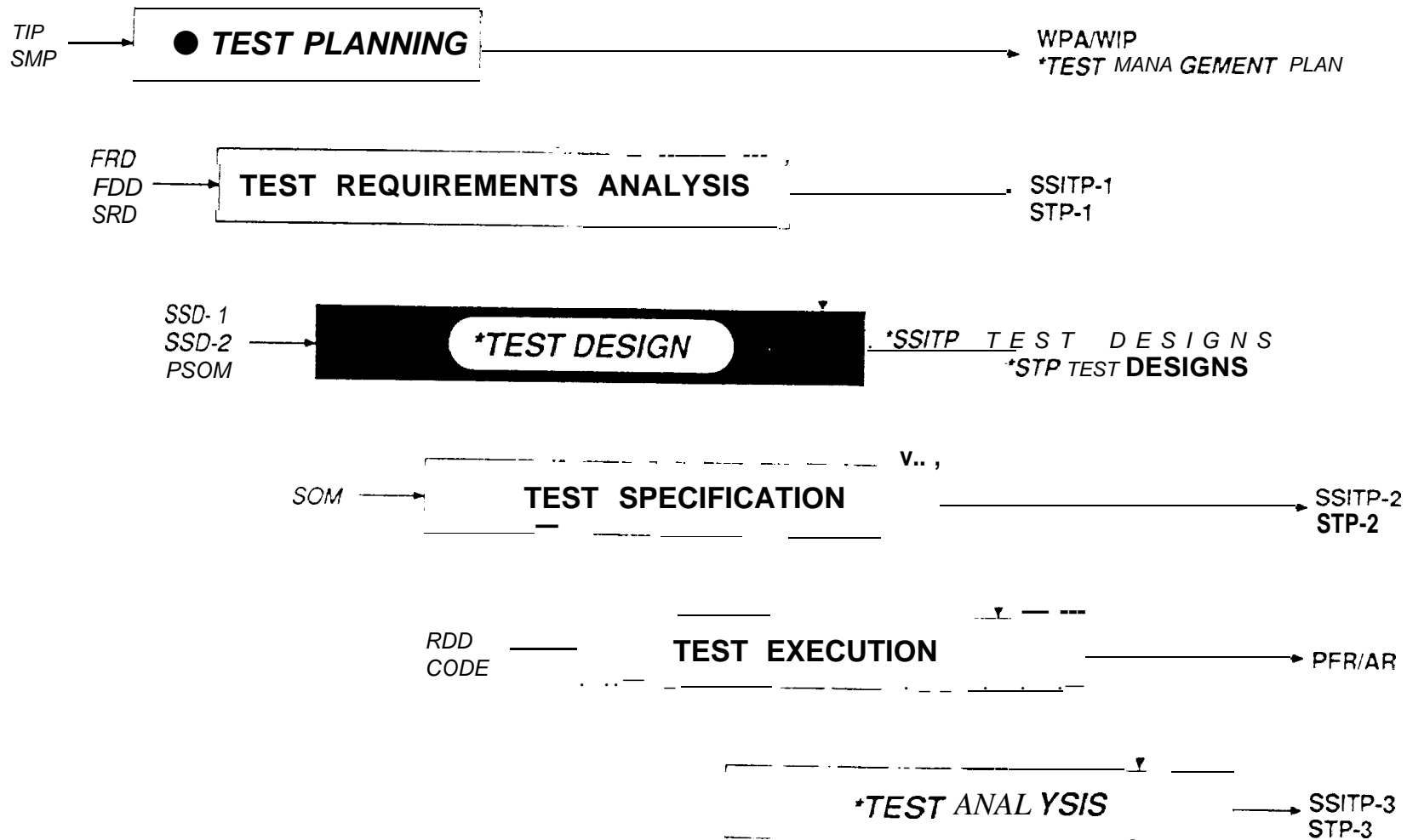
## 2. Problem Definition

### 2.1 Testing Life Cycle

In the field of software engineering, we are often faced with the challenge of creating an integrated, working system based on inadequate and meager requirements. The waterfall lifecycle for software development - where requirements are systematically refined, architectural detail design established, code written, and then the system tested, has become an accepted method for dealing with the ambiguities and vagueness of the original requirements. The testing portion of this development lifecycle, however, is not so clearly defined or widely accepted.

Figure 2-1 depicts the testing life cycle used in some of the development projects at Jet Propulsion Laboratory, operated for NASA by the California Institute of Technology. This life cycle is described in JPL's Software Management Standard, D-4000 [2], and is similar to the software development standard adopted by the Department of Defense, DOD-STD-2167A [3]. Table 2-1 defines the acronyms used in Figure 2-1, along with the title of the document it stands for, the phase in the testing lifecycle during which it is used or produced, and finally, whether a description of its contents is in the Standards [2] and [3].

Figure 2-1 Testing Life Cycle



LEGEND: D-4000 INPUT D-4000 PHASE OR OUTPUT 'PHASE OR OUTPUT NOT DEFINED IN D-4000

Table ?-1 Testing Life Cycle Inputs and Outputs

Acronym	Document Title	Testing Life Cycle Phase	Standards Exist?
D-4000	JPL's Software Management Standard	all	yes
TIP	Task Implementation Plan	Test Planning	yes
SMP	Software Management Plan	Test Planning	yes
WPA/WIP	Work Package Agreement/ Work Implementation Plan	Test Planning	yes
Test Management Plan	Test Management Plan for defining procedures of complete testing cycle	Test Planning	no
FRD	Functional Requirements Document	Test Requirements Analysis	yes
FDD	Functional Design Document	Test Requirements Analysis	yes
SRD	Software Requirements Document	Test Requirements Analysis	yes
SSITP-1	SubSystem Integration anti Test Plan - 1, requirements	Test Requirements Analysis	yes
STP-1	Software Test Plan -1, requirements	Test Requirements Analysis	yes
SSD-1	Software Specification Document -1, architectural design	Test Design	yes
SSD-2	Soft-ware Specification Document -2, detail design	Test Design	yes
PSOM	Preliminary Software Operator's Manual	Test Design	yes
<b>SSITP Test Designs</b>	SubSystem Integration and Test Plan - Test Designs for <b>SSITP-1</b>	Test Design	no
STP Test Designs	Software Test Plan - Test Designs for <b>STP-1</b>	Test Design	no
SOM	Software Operator's Manual	Test Specification	yes
SSITP-2	SubSystem Integration and Test Plan -2, detailed procedures	Test Specification	yes
STP-2	Software Test Plan -2, detailed procedures	Test Specification	yes
RDD	Release Description Document	Test Execution	yes
PFR/AR	Problem Failure Report/ Anomaly Report	Test Execution	yes
SSITP-3	SubSystem Integration and Test Plan -3, report	Test Analysis	yes
STP-3	Software Test Plan -3, report	Test Analysis	yes

As noted in both Figure 2-1 and Table 2-1, a key component is missing from the standards - a definition of the inputs, outputs and purpose of the Test Design Phase. This gap between the requirements for testing, produced in the Test Requirements Analysis phase, and the detailed test procedures, produced in the Test Specification phase, is the phase during which the category-partition method can be most useful. The test design phase is explored in detail in the following section.

## 2.2 Test Design Phase

The dictionary definition of the word "design" is to conceive and to devise for a specific purpose. During the Test Design Phase, the "specific purpose" that the test engineer is concerned with is meeting the test objectives and requirements determined in the Test Requirements Analysis Phase; what the test engineer is trying "to conceive and to devise" are the necessary and sufficient ways of validating the functional and performance requirements of the entire system. Therefore, the purpose of the Test Design Phase is to conceive and specify the environmental and system attributes that verify requirements and meet test objectives for each test requirement in the test plan, and for each requirement in the functional and software requirements documents.

Based on this definition of the purpose, the inputs to this phase are relatively simple to identify. They are:

- a) the test objectives as documented in the SubSystem Integration and Test Plan and/or the Software Test Plan;
- b) the functional and performance requirements and system design as documented in the Software Specification Documents or the Functional and Software Requirements Documents; and
- c) any other pertinent design or requirements information that may be available, such as Interface Agreements and the Preliminary Software Operators Manual.

The outputs from the Test Design Phase to the next phase, however, are not so easy to identify. The products we are trying to develop are ways to validate requirements, which will be referred to as test designs. These designs are not expected to be test procedures specified to enough detail to be run by another test engineer or possibly a third party; the test procedures written to that detail will eventually be written in the subsequent phase of the testing lifecycle, the Test Specification Phase. The test designs can have some ambiguity in the sequence of steps, the testing range of certain parameters, or the actual testing steps themselves.

Additionally, each test design should directly imply or specify a group of test cases. The test cases should have specific values for environmental and/or system parameters that have an effect on how the system under test will behave. Each of the test cases should also include the expected response or behavior of the system.

With this in mind, the outputs of the Test Design Phase can be stated as:

- a) test design specifications that:
  - 1) are traceable to test objectives and functional and/or software requirements;
  - 2) directly imply or specify a group of test cases that can be individually executed but share the same set-up procedures;
  - 3) identify the environmental and system features that are to be set or observed to control and determine the behavior of the system;
  - 4) pass criteria for the group of test cases; and
- b) test cases that specify:
  - 1) the environmental and/or system parameters and system states that should exist before the test case is executed;
  - 2) the test action or step to be taken to initiate the system behavior; and
  - 3) the expected system behavior after the action has been taken.

In the following sections, methods for determining the test designs and for automatically producing the documentation for the test cases are presented.

## 3. Method of Solution

There are many ways to create test designs that meet the needs of a certain project. Four of these methods are discussed below: the representative set method, the ad-hoc method, the all-permutations method, and finally the category-partition method.

A common method for determining the number and contents of the test designs and test cases that should be transformed into test procedures is selecting a representative set of normal conditions and parameters that prove that the system works and meets requirements. On a project using this method, the emphasis will be on demonstrating that the system works rather than testing the system to detect failures, but the repeatability of the test procedures and the traceability to the requirements being tested is generally good.

On projects that are particularly short of time, money, and personnel, the test design phase is almost totally skipped. In this case, the test design method can be characterized as ad-hoc. The ad-hoc test case selection is particularly prone to missing important aspects of the system behavior that could help determine where the problems are. The emphasis on a project using this method is almost always on getting the system "out-the-door". Traceability to requirements is often poor. And most devastating of all, test repeatability is sacrificed; when a failure eventually occurs and the problem solved, it is very difficult to verify that the fix was correct because the conditions that caused the failure cannot be repeated.

Though not often seen, another method for selecting test designs and cases is a brute force method of analyzing all permutations of system parameter values. With this method, the test designs and cases are easily traced to requirements and test objectives, but it takes a lot of time and effort to analyze each permutation and decide which ones are valid and which ones are meaningless. This method allows the test engineer to find test cases that lie on the extreme boundary of the valid input space, and therefore is good for error detection.

A recommended method for determining test designs is the category-partition method [5]. This method combines the benefits of choosing normal cases with the error exposing properties of the all-permutations method. Traceability can be maintained quite easily by creating a test design for each test objective in the test plan. Using an automatic tool to create the test cases based on the test design, the subsequent effort to transform the test cases into test procedures is simplified. The method allows the rapid elimination of undesired test cases from consideration, and easy review of test designs by peer groups.

Section 3.2 discusses the category-partition method in general, and is followed by section 4 which presents the Test Representation language (TRL) that can be used to implement the method and produce the test cases using the TRL tool.

### 3.2 Category-Partition Method

#### 3.2.1 Background

The category-partition method was first presented by Ostrand and Balcer in 1988 [5]. A follow-on paper in 1989 [1] discussed a test specification language and a tool for the automatic generation of test scripts that could be compiled and executed in the test environment that they had set up at Siemens Corporate Research. As pointed out in these two papers, the category-partition method is a way of analyzing the functional and software requirements of a system in order to determine test cases to be run. The method relies exclusively on the test engineers' reading of the requirements and design documents and their judgement of exactly which test cases should be selected for procedure development. If a formal requirements specification language is used to document the requirements and design, other methods may be more useful, such as the ones described in the paper by Richardson, et. al. [6]. However, it is not often that the test engineer is presented with a functional requirements document or a software requirements document that is written this formally. Therefore, a structured method, such as the category-partition method, is needed to provide a systematic approach to developing test specifications from informal representations of the required system behavior.

The following sections discuss the steps in the category-partition method. The Steps have been organized differently than the procedure discussed in the primary references, [5] and [1]. The organization of steps presented below has proven useful in communicating the method to the test engineers on our projects.

### 3.2.2 Steps in the Category-Partition Method

The category-partition method consists of four steps:

- 1 Functional Decomposition;
- 2 Category Analysis;
- 3 Partition Value Analysis; and
- 4 Partition Constraint Analysis.

Each of these steps are discussed in the following sections.

#### 3.2.2.1 Step 1: Functional Decomposition

The first step in the category-partition method is functional decomposition. The purpose of this step is to decompose the specification and/or requirements into functional units that can be tested independently. A secondary purpose of this step is to identify the parameters that affect the behavior of the system for each functional unit.

The requirement space is subdivided into subgroups, which may or may not overlap in some aspect. Each subgroup clearly identifies the requirements being tested and the input, output, and environmental parameters that affect how the system meets the requirements. The types of parameters that should be considered are: user inputs, inputs from external interfaces, environmental inputs, outputs to another (observable) portion of the system, outputs to a user or external interface, outputs to the environment or state of the system, or maybe even the sequence of events. Note that there will be times when some of the parameters are not explicitly stated in the requirements specification, and therefore implicit parameters will have to be determined.

For an example, assume the following requirement specification has been decomposed from the requirement space:

**Sort an integer array either in ascending or descending order,**

The parameters mentioned explicitly in this requirements statement are the array, and **an indication of sort order**. Implicitly, however, **the** result of the sort operation is also a parameter for this requirement.

The next step of the procedure is to further- analyze the parameters identified and determine the characteristics, or categories, of the parameters that affect program **or** system execution.

#### 3.2.2.2 Step 2: Category Analysis

The second step in the category-partition method is category analysis. The work done in the previous step, identifying functions] units and explicit and implicit parameters, is carried further by determining the properties or sub-properties of the parameters that would make the system behave in different\_ ways. The test engineer should analyze the requirements and determine the features or categories of each parameter and how the system may behave if the category were to vary its value. If the parameter undergoing refinement were a data-item, then categories of this data-item may be any of its attributes, such as type, *size*, value, units, frequency of change, or source.

Choosing the "array" from the example in step 1 for further refinement, the categories that may be derived from the specification are "array **size**", "the values **in** the array", and, because the functional unit is a sorting function, "the arrangement of the values **in** the array" .

As can be seen, the original requirement statement said nothing about the valid range

of "array size". This step, along with the next one, tend to point out deficiencies in the requirements specification. The test engineer will have to work closely with the author of the requirements and the designers in order to resolve the ambiguities and uncertainties that surface from this analysis.

### 3.2.2.3 Step 3: Partition Value Analysis

After all the categories for the parameters of the functional unit have been determined, the next step is to partition each category's range space into mutually exclusive values that the category can assume. In choosing partition values, the focus should be on error exposing values. The discussion on boundary value testing in Myers' book [4], and revealing subdomains in the paper by Weyuker and Ostrand [7] should prove useful as references.

The partition values should include all possible kinds of values, especially the ones that will maximize error detection. Important values to look for are boundary values, extremal and non-extremal values, values that represent special cases or interactions, and valid and invalid values.

Returning to the example and using the category "array size" for illustration, the five partition values are:

- 1) 0
- 2) 1
- 3) 2 to the Upper Bound minus 1
- 4) Upper Bound
- 5) greater than the Upper Bound

It can be seen that "0" and "greater than the Upper Bound" represent error conditions that the sort. function will have to process, while "1", and "Upper Bound" represent special cases or boundary values. The reason the all the values between "2" and the "Upper Bound minus 1" (inclusive) have been grouped together is because the sorting function is expected to behave the same in this range; an error in processing that occurs for a particular value in this range should occur for all the values in this range. It is left up to the Test Specification Phase of the testing lifecycle to determine the exact, or random, values that should be used to verify this partition in the test procedure.

The fact that two of the five values in this example have already been identified as being representative of error conditions gives us a head start on the next step of the category-partition method.

### 3.2.2.4 Step 4: Partition Constraint Analysis

The purpose of this final step is to refine the test design specification so that only the technically effective and economically feasible test cases are implied. There are three types of constraints defined in the category partition method as described in [5]: Errors, Limits, and Conditions.

An Error Constraint applied to a partition value is used to indicate that the partition value represents an exception state that the system under test should not-e and report without processing any further. partition values of this type need to be tested in one test case but no more, due to the way exceptions are usually handled. An example of partition values that should have Error Constraints are "0" and "greater than the Upper Bound" in the category of "array size".

A Limit Constraint is for limiting the number of times a partition value will be used in the resulting test cases. Limit Constraints can be applied to a test design in order to control the actual number of test cases implied. When economic feasibility, as in restricted time and resources, is a factor in the test execution, the Limit Constraint will help the test engineer to eliminate some of the test cases that seem redundant. In the example we have been following, the test engineer may want to limit the number of times that an "array size" of "Upper Bound" is used.

The remaining type of constraint is the Conditional Constraint. Determining these types of constraints is where the majority of the intellectual effort is spent. This

part of the analysis specifies which partition values from one category can be used with the partition values of another category. Conditional Constraints are specified in pairs: pre-conditions and post-conditions. Pre-conditions are states or conditions that must co-occur for a particular partition value to be used in a test case; post-conditions are the states or conditions that are set when a partition value is used. To illustrate their use, a slightly more involved example should be discussed.

Starting with the category of "array **size**" and the partitions determined in the previous step, we analyze the types of conditions that are expressed by each partition value. It can be seen that the values represent 3 separate conditions:

- a) "error occurs" (for partition values of "0" and "**greater than Upper Bound**") ;
- b) "size is normal" (for partition values of "2 to Upper Bound **minus 1**" and for "**Upper Bound**"); and
- c) "size represents a degenerate array" (for an array size of "1") .

Clearly, if everything else is set appropriately, the valid partition values of the category "**result**" will be dependent on these conditions. Let's assume the following **4 partition values were identified in Step 3 for the "result" category: "error notification", "array unchanged", "array in ascending order", and "array in descending order"**. A pre-condition for the result "error notification" is that the post-condition "error occurs" has been set. For the values of "array in ascending order" and "array in descending order" the post-condition of "size is normal" must, have been set before these values could be used in a valid test case. The "result." of "**array unchanged**" could possibly be a result of many conditions, one of which is that the "**array size**" is "1", where the "size represents a degenerate array".

### 3.2.3 Example Application of Category-Partition Method

Table 3-1 provides the results of the method applied to the example that has been discussed Throughout the previous sections of this article.

Table 3-1 Example Formal Test Design: Application of Category-Partition Method

Functional Unit: Sort an integer array either in ascending or descending order.			
Categories	Partitions Values	Partition Constraints: Post-Conditions	Partition Constraints: Pre-Conditions
1. array size	0 (array unspecified)	SET "error occurs"	
	1 (degenerate array)	SET "size represents a	
	2 to upper bound minus 1	SET "size is normal"	
	Upper Bound	SET "size is normal"	
	greater than Upper Bound	SET "error occurs"	
2. array values	all zero	SET "values identical"	IF "size is normal"
	all positive values	SET "not identical"	IF "size is normal"
	all negative values	SET "not identical"	IF "size is normal"
	mixed positive, negative,	SET "not identical"	IF "size is normal"
	don't care		IF "error occurs" or IF "size represents a degenerate array"
3. value arrangement	minimum value before maximum value		IF "not identical"
	maximum value before		IF "not identical"
	don't care		IF "values identical" or IF
4. sort order	unspecified	SET "error occurs"	IF "size is normal"
	ascending order	SET "ascending order"	IF "not identical"
	descending order	SET "descending order"	IF "not identical"
	don't care		IF "values identical" or IF

Functional Unit: Sort an integer array either in ascending or descending order.			
Categories	Partitions Values	Partition Constraints: Post-Conditions	Partition Constraints: Pre-Conditions
5. result	error notification		IF "error occurs"
	array unchanged		IF "values identical"
	array in ascending order		IF "*ascending order"
	array in descending order		IF "descending order"

#### 4. Test Representation Language (TRL)

The Test Representation Language (TRL) was developed to implement the category-partition method. When used during the Test Design E'base of the testing lifecycle, the TRL files will form concise and uniform representations of the test designs for the functional testing of the system.

The TRL tool that implements the TRL language processes the ASCII formatted TRL files and produces ASCII formatted result files that document the individual test cases implied by the test design. The TRL tool documents the description, categories, and partition values to be used in each test case. Each TRL file is created and modified with an ASCII editor and therefore can be easily modified to adapt to changes in functional specifications. The resulting test case descriptions can be used during engineering tests of the system under test to verify preliminary procedures and functions while work continues in the Test Specification E'base, transforming the test case descriptions into formal detailed test procedures.

The TRL tool was written in the "C" programming language and can be ported to any platform; the SUN/SPARC and DOS environments are the computer platforms on which it currently runs. This tool differs from the one described in reference [1], in that the TRL tool is a general permutation control language that can be used in any environment; the output of TRL the tool is ASCII files that can be used for documentation rather than an executable test script.

##### 4.1 TRL Language Definition

The Test Representation Language (TRL) provides a way to describe many test cases with one TRL File. The language consists of 1 comment character, 11 keywords, 2 field demarkation characters, a logical AND character, and a logical NOT character. The processing rules for the keywords, comments, and fields appears in the following sections and a summary of the Test Representation Language appears in Figures 4-1.

##### 4.1.1 Special Characters

There are five special characters in TRL character set:

- 1) comment character = asterisk (\*) ;
- 2) start field character = open bracket ( [ ) ;
- 3) end field character = close bracket ( ] ) ;
- 4) logical AND character = comma ( , ) ; and
- 5) logical NOT character = exclamation point ( ! ) .

The asterisk is for initiating a comment line, which is a line defined by the comment character appearing as the first non-white-space character on a line in the TRL file.

The start field character and end field character are for specifying the beginning and ending of a partition constraint field. Partition constraint fields are discussed in detail in section 4.1.3.

The logical AND character and the logical NOT character are for specifying logical relation inside a partition value constraint field that is used for setting conditional constraints. See section 4.1.3.1 and 4.1.3.2 for examples of their use.

##### 4.1.2 Line Keywords

There are two types of keywords in the Test Representation Language: line-keywords and field-keywords. To be recognized as valid, the line-keywords should be the first word on a line. These keywords are used to initiate a description of the test designs (DESCRIPTION), indicate the beginning of the categories and partitions (PARAMETERS), indicate a certain type of category (TYPE), specify the name of a category (NAME), set error message text (MESSAGE), and to indicate the start of the block that describes the partition value and constrains of each category (SAMPLES). The line-keywords are, respectively:

DESCRIPTION, PARAMETERS, TYPE, NAME, SAMPLES, MESSAGE.

Figure 4-1 Test Representation Language (TRL) Summary

<u>Character or Keyword</u>	<u>Purpose and/or Usage</u>
<b>*</b>	Indicates a comment line.
<b>DESCRIPTION</b>	Indicates the starting of a description block that will be included in test cases.
<b>PARAMETERS</b>	Indicates the beginning of parameter specifications.
<b>NAME</b>	Specifies the name of a parameter or category.
<b>TYPE</b>	Indicates the type of category.
<b>SAMPLES</b>	Indicates beginning of a samples block defining the partition values and constraints,
<b>[</b>	Beginning of sample value constraint field.
<b>]</b>	End of sample value constraint field.
<b>IF</b>	Field identifier indicating that pre-condition constraints are listed in the current field. Comma (,) is used for logical AND, exclamation (!) for logical NOT.
<b>SET</b>	Field identifier indicating that post-condition constraints are listed in the current field.
<b>LIMIT m</b>	Field identifier indicating that the number of test cases involving this partition value should be limited to m. If m is unspecified, the limit is one test case.
<b>LABEL</b>	Field identifier indicating that the specified label should be listed for this partition value.
<b>ERROR n</b>	Field identifier indicating that the sample value is an error exit. The error can be specified using the optional n.
<b>MESSAGE n</b>	Indicates that a message block follows corresponding to the errors in the partition values. The message number can be specified using the optional n.
<b>Command Line Options</b>	For performing "count only," writing results into separate files, and for including pre/post conditions in output.

#### 4.1.2.1 DESCRIPTION Keyword

The purpose of the DESCRIPTION keyword is to indicate the start of a description block. The description block will be printed at the top of each test case results file. It is recommended that this contain the requirements that the TRI File addresses, the TRI file name, author, date, and a description of the general test set.

The description block starts at the first non-comment line following the DESCRIPTION keyword; the block ends at the first comment line or keyword line after the description block has been started. It may contain blank lines but no keyword lines or comment lines.

If no DESCRIPTION keyword is found in the TRI file before the PARAMETERS keyword, or there was no description block then the default description, which is the name of the TRI file that contains the test design, will be printed at the top of each test case results file.

```
Example 1:
  DESCRIPTION
    *
    *
    Requirements: 1.1, 1.2, 1.3
    TRI File:      electrical environment.TRI
    (results in file: electrical environment.RES)

    Last Modified: 5/5/91

    * This comment line ends the block
```

```
Example 2:
  DESCRIPTION
    * empty description
  PARAMETERS
```

#### 4.1.2.2 PARAMETERS Keyword

The purpose of the PARAMETERS keyword is to identify the beginning of the test representation and the corresponding categories and partitions. It must appear before all of the line-keywords, except DESCRIPTION. It can only appear once in the TRI file. All other keywords must appear after PARAMETERS in TRI File.

```
Example 1 :
  *
  PARAMETERS
  *
```

#### 4.1.2.3 TYPE Keyword

The purpose of the TYPE keyword is to set the current type of the subsequent parameters and categories. The type associated with each parameter is printed along with the partition values in the test case files. The TYPE keyword must be followed by the new default type specification on the same line. The specification can have alpha-numeric characters, embedded spaces, and punctuation characters.

If there is no specification (i.e. only white-space appears after the TYPE keyword), then the previous TYPE setting will be used. The default TYPE setting is "Parameter".

```
Example 1:
  TYPE      Input Parameter
```

Examples 2, 3, 4, and 5:

TYPE	Output Parameter
TYPE	State Parameter
TYPE	Environment Parameter
TYPE	

#### 4.1.2.4 NAME keyword

The purpose of the NAME keyword is to set the name of the current parameter or category. The NAME parameter indicates the beginning of a new category specification. The category specification consists of a NAME; setting and the samples or partition values associated with it as initiated by the SAMPLES keyword.

The NAME keyword must be followed by the name specification on the same line. The specification can have alpha-numeric character, embedded spaces, and punctuation characters. If there is no name specification (i.e. only white-space appears after the NAME keyword), then the default parameter name is used.

If there are no samples following the NAME keyword line prior to the end-of-file or the next NAME keyword line, then TRL processing stops.

See the SAMPLES keyword description, section 4.1.2.5, for information on SAMPLES.

Example 1 :

```
*
NAME      sort order
*
SAMPLES
...
```

Example 2:

```
NAME Anything can follow this (even punctuation. !)
SAMPLES
...
```

#### 4.1.2.5 SAMPLES Keyword

As mentioned above, the category specification consists of a NAME setting and the samples or partition values associated with it as initiated by the SAMPLES keyword. This keyword begins the processing of the partition values and the partition value constraints for the category indicated by the NAME keyword.

The associated partition value block starts with the next non-comment line and ends with the subsequent blank line, comment line, or a line with another keyword on it. An empty block will cause TRL processing to stop.

Details of the partition value constraint analysis can be found in the descriptions of the SET, IF, ERROR, LIMIT, and LABEL keywords, in section 4.1.3.

In order to be recognized properly as partition value text, the partition value text cannot start with the comment character, "\*", and cannot contain any start or end field characters, "[" or "]".

Example 1 :

```
*
NAME      sort order
*
SAMPLES
*          3 samples for "sort order"
*
```

```

        ascending
        descending
        don't care
    * (this comment line ends the value line block)

```

Example 2:

```

NAME Anything can follow this (even punctuation.!)
SAMPLES
    1
    2
    3

    * (the blank line, above, ends the value line block)

```

#### 4.1.2.6 MESSAGE Keyword

The MESSAGE keyword is used to start the processing of a message block of lines. The messages in the message blocks are printed when a partition value is used in a test case that has an ERROR indication.

The format used for the Message Keyword is: MESSAGE n. The optional "n" is either blank or an integer. The MESSAGE keyword indicates that the next line or lines should be entered in the "n"th position of the error message list. If "n" is unspecified, then the next available message number will be assigned.

The message block itself starts on the line following the line with the MESSAGE keyword on it, and is terminated by a blank line, comment line, the end of the file, or a line with another keyword.

No special processing for the message line is performed beyond keeping track of the error message list and its associated numbers.

Example 1:

```

    *
    MESSAGE
        Error message number is the (next_ available)
        Error message number is the (next available)+1
    *
    * the comment line above terminates message processing

```

Example 2:

```

    *
    MESSAGE 6
        Error message #6
        Error message #7
        Error message #8

    * blank line terminated the message processing.

```

#### 4.1.3 Field Keywords

The field keywords are used in the partition value constraint fields to either describe a partition value (LABEL), or to specify the constraints determined during Step 4 of the category-partition method. The field-keywords for setting labels and constraints are:

SET, IF, LIMIT, ERROR, and LABEL.

A partition value constraint field is associated with a particular partition value by its physical location in the partition value block. A line in this block consists of the partition value text followed by zero or more constraint fields. The constraint fields can extend beyond the physical line of the TRI file, but the partition value text cannot. Partition value constraint fields are started by the start field characters, "[", and ended by the end field characters, "]".

As previously mentioned, partition value text cannot start with the comment character, "\*", anti cannot contain any start or end field characters, "[" or "]" .

The field keywords are described in the following subsections.

#### 4.1.3.1 SET Keyword

The purpose of the SET field keyword is to set post-conditions. As explained in section 3.2.2.4, post-conditions (and pre-conditions) arise during the partition constraint analysis step of the category-partition method. For a particular partition value, the post-conditions specified in the SET field will be set if the partition value is used in a test case. A corresponding pre-condition, as determined by an IF field, is used to control which partition values from different categories are validly used together in a test case.

The SET keyword must be the first word following the "[" character. The post-conditions themselves are one or more words that can consist of alphanumeric characters plus "'", or '-'. The SET field is terminated by a "]" or the end of the line.

The logical AND character, a comma ",", should separate post-conditions when more than one post-condition will be set.

Example 1:

```

*
NAME          sort order
*
SAMPLES
*           3 samples for "sort order"
*
      ascending          [ IF size ok, ruin_max]
                        [ SET ascend, normal]
      descending         [IF size_ok, min_max ]
                        [SET descend, normal]
      don't care         [IF !normal]
                        [SET dont_care]
*
* EXPLANATION:
*
* The "ascending" anti "descending" values will be
* used only when the post-conditions "size_ok" and
* "min_max" have been SET by the use of other partition
* values.
* The use of "ascending" will SET the "ascend" anti
* "normal" post-conditions. The use of "descending"
* will SET the "descend" and "normal" post-
* conditions.
*
* The "don't care" value will only be used if the
* post-condition "normal" is NOT SET by any other
* sample value. When valid, the use of this partition
* value will SET the "dont care" post-condition.
*

```

#### 4.1.3.2 IF Keyword

The purpose of the IF field keyword is to set a pre-condition for the use of the associated partition value. The IF keyword must be the first word following the "[" character. The pre-conditions themselves are one or more words that can consist of alphanumeric character plus "'", or '-'. The IF field is terminated by a "]" .

The logical AND character, a comma ",", can be used to separate a condition when multiple pre-conditions are necessary before a partition value can be used in a test case .

The logical NOT character, an exclamation point "!", can precede a pre-condition to indicate that a particular pre-condition cannot exist if the partition value is to be used in a test case.

The partition value text will only be used in a resulting test case if the pre-conditions specified in the IF field are met by the post conditions that are set (using the SET field keyword) by the other values in the resulting test case.

Example 1 :

```

*
NAME          sort order
*
SAMPLES
*          3 samples for "sort order"
*
          ascending          [ IF size ok, min max]
          descending          [IF size ok, min max ]
          don't care          [IF !normal]
*
* EXPLANATION:
*
* the "ascending" anti "descending" values will be
* used only when the post-conditions "size ok" and
* "rein max" have been SET in other sample values.
* The "don't care" value will only be used if the
* post-condition "normal" is NOT SET by any other
* sample value.
*

```

#### 4.1.3.3 LIMIT Keyword

The Limit field keyword is used to set a limit on the number of test cases in which the associated partition value can be used. The format of the field when LIMIT is used is: [LIMIT n], where "n" is either blank or an integer. If "n" is specified then a maximum of this number of iterations will be allowed to be used in the resulting test cases. The number "n" starts with the first numeric after the LIMIT keyword and ends with the first non-numeric following that.

The LIMIT keyword must be the first word following the "[" character. The LIMIT keyword indicates that the current partition value is limited to the number of specified iterations, where "n" is the maximum. If "n" is not specified then "n" is assumed to be one. The LIMIT field is terminated by a "]".

Example 1 :

```

*
NAME          sort order
*
SAMPLES
*          3 samples for "sort order"
*
          ascending          [ IF size ok, min max]
                              [ SET ascend, normal]
                              [LIMIT 10]
          descending          [IF size ok, min max ]
                              [SET descend, normal]
          don't care          [IF !normal]
                              [SET dent-care]
*
*
* EXPLANATION:
*
* The "Ascending" and "descending" values will be
* used only when the post-conditions "size ok" and
* "min max" have been set (using the SET field keyword)
* by other categories in the test design.

```

```

* The use of "ascending" will SET the "ascend" and
* "normal" post-conditions. The "ascending"
* partition value will be used a maximum of 10 times.
*
* The use of "descending" will SET the "descend"
* and "normal" post-conditions. There is no LIMIT
* on the number of times the "descending" partition
* value can be used in the test cases.
*
* The "don't care" value will only be used if the
* post-condition "normal" is NOT SET by any other
* sample value; it's use will SET the dont care
* post-condition. There is no LIMIT
* on the number of times the "don't care" partition
* value can be used in the test cases.
*

```

#### 4.1.3.4 ERROR Keyword

The purpose of the ERROR field keyword is to indicate that the current partition value represents an error condition. The partition value, therefore will only be included one time in the resulting test cases. The ERROR keyword must be the first word following the "[" character. The ERROR keyword indicates that the current partition value should raise an exception in the system under test and a specific error message **should be observable**.

The format of the field when ERROR is specified is: ERROR n. The optional "n" is either blank or an integer. If "n" is specified then the "n"th message in the error message list will be printed along with the use of this partition value! in a test case. The message list is determined by the MESSAGE keyword, as described in section 4.1.2.6.

The number- "n" starts with the first numeric after the ERROR field keyword and ends with the first non-numeric following that. If "n" is not specified, then the errors will be numbered as they are encountered in the TRL file.

Example 1:

```

*
NAME          sort order
*
SAMPLES
*           4 samples for "sort order"
*
      ascending          [ IF size_ok, min_max ]
                          [ SET ascend, normal ]
                          [ LIMIT 10 ]
      descending          [ IF size_ok, min_max ]
                          [ SET descend, normal ]
      unspecified          [ IF all_normal ]
                          [ SET dont_care ]
                          [ ERROR 5 ]
      don't care          [ IF !normal ]
                          [ SET dont_care ]
*
MESSAGE 5
      Error Message #5: sort order was not set
*
*
* KXPLANATION:
*
* The "ascending" and "descending" values will be
* used only when the post_ conditions "size ok" and
* "min_max" have been SET by the use of another
* partition value in a different category.
* The use of "ascending" will SET the "ascend" and

```

```

* "normal" post-conditions . Also, the "ascending"
* sample value will be used a maximum of 10 times.
*
* The use of "descending" will SET the "descend"
* and "normal post-conditions . There is no LIMIT
* on the number of times the "descending" sample
* value! can be used in the test cases.
*
* The "unspecified" partition value will be used in a
* test case only if the "all normal" post-conditions
* was set. The "unspecified" partition value will only
* be used once and the 5th error message in the error
* message list will be printed when this partition
* value is used. The "dont care" post condition will be
* set .
*
* The "don't care" value will only be used if the
* post-condition "normal" is NOT SET by the use of
* another partition value; if "don't care" is used, then
* the dont care post-condition will get SET.
* 'I'here is no LIMIT on the number
* of times the "don't care" sample value can be used
* in the test cases.
*

```

#### 4.1.3.5 LABEL Keyword

The purpose of the LABEL field keyword is to indicate that the associated partition value should be labeled with the specified text when it is used in a resulting test case . The format of the field when LABEL is used is: [LABEL label text], where label text is any string of characters except "]".

The LABEL keyword must be the first word following the "[" character. The LABEL keyword indicates that the following label text is the label for the current partition value. If "label text" is specified then the "label text" will be printed along with the use of this sample. in a test case. Otherwise, the default label "valid" will be used.

Example 1 :

```

*
NAME          sort order
*
SAMPLES
*           4 samples for "sort order"
*
      ascending      [ IF size_ok, min_max ]
                      [ SET ascend, normal ]
                      [ LIMIT 10 ]
                      [ LABEL valid sort order ]
      descending      IF size_ok, min_max ]
                      [ SET descend, normal ]
                      [ LABEL valid sort order ]
      unspecified      IF all normal ]
                      [ SET dont_care ]
                      [ ERROR 5 ]
      don't care      [ IF !normal ]
                      [ SET dont_care ]
                      [ LABEL any valid sort order ]
*
MESSAGE 5
      Error Message #5: sort order was not set
*
* HXP1,ANATION:
*
* The "ascending" and "descending" values will be

```

\* used only when the post-conditions "size ok" and  
 \* "min max" have been SET by the use of other values in  
 \* one or more different categories specified in the  
 \* TRI file.  
 \* The use of "ascending" will SET the "ascend" and  
 \* "normal " post-conditions. Also, the "ascending"  
 \* sample value will be used a maximum of 10 times.  
 \*  
 \* The use of "descending" will SET the "descend"  
 \* and "normal " post-conditions. There is no LIMIT  
 \* on the number of times the "descending" sample  
 \* value can be used in the test cases.  
 \*  
 \* Both "ascending" and "descending" will have the  
 \* label "valid sort order" listed in the resulting  
 \* test cases when these partition values are used.  
 \*  
 \* The "unspecified" sample value will be used in a  
 \* test case only if the "all normal" post-conditions  
 \* was set. This sample value will only be used once  
 \* and the 5th error message in the message list will  
 \* be printed when this sample value is used. The  
 \* "don't care" post condition will be set.  
 \*  
 \* The "don't care" value will only be used if the  
 \* post-condition "normal" is NOT SET by any other  
 \* sample value; its use will SET the don't care  
 \* post-condition. There is no LIMIT on the number  
 \* of times the "don't care" sample value can be used  
 \* in the test cases. This sample value will have  
 \* the label "any valid sort order" when it is used.  
 \*

#### 4.2 Example Application of Category-Partition Method with TRI

In this section, the same example from section 3.2.2 will be discussed, but this time, the Test Representation language (TRI) will be used. To avoid confusion, the procedures for creating a test design using TRI are referred to as stages, and the procedures for implementing the category-partition method are referred to as steps. These stages will be performed for each functional unit and/or test objective in the system under test.

##### 4.2.1 TRI STAGE 1: Unconstrained Representation

The first stage in the TRI procedure is to create an unconstrained representation of the test design. This is accomplished by performing the first 3 steps in the category-partition method:

- Step 1: Functional Decomposition (section 3.2.2.1) ;
- Step 2: Category Analysis (section 3.2.2.2); anti
- Step 3: Partition Value Analysis (section 3.2.2.3) .

As for creating a TRI File, the following TRI keywords and information should be created:

- a) **DESCRIPTION** Keyword and the **description block**. Create a description block that contains the requirements to be tested, the pass criteria to be used, and any other pertinent information to the test design.
- b) **PARAMETERS** Keyword. Start the parameter specification block.
- c) **TYPE** Keywords and **NAME** Keywords. For each type of parameter and category identified in Step 2 of the category-partition method, create a TYPE and NAME specification in the TRI file.
- d) **SAMPLES** Keywords and the partition values. For each category, add in the unconstrained partition values that the category can assume during a test.

Example:

```
*
DESCRIPTION
*
Functional Unit:  Sort an integer array either in ascending or
                  descending order.
*
PARAMETERS
*
TYPE  Input-Categories for Parameter: Array

      NAME  array size
      SAMPLES
            0
            1
            2 to Upper Bound minus 1
            Upper Bound
            greater than Upper Bound

      NAME  array values
      SAMPLES
            all 0's
            all the same but not 0
            all negative
            all positive
            mixed +/-/0
            don't care

      NAME  value arrangement
      SAMPLES
            minimum before maximum
            maximum before minimum
            don't care
*
TYPE  input--Parameter: Sort Order

      NAME  sort order
      SAMPLES
            ascending
            descending
            unspecified
            don't care
*
TYPE  Output to program or change in state

      NAME  result
      SAMPLES
            error notification
            array unchanged
            array in ascending order
            array in descending order
*
* end of file
*
```

Note that this example with the unconstrained representation would produce 1440 test cases.

#### 4.2.2 TRL STAGE 2: Error Constrained Representation

The second stage of this process is to add in the error indicators and the! message descriptions. This corresponds to a portion of the fourth step in the category-

partition method: the partition constraint analysis.

The following keywords and information should be added to the TRI file:

- a) **ERROR field keywords.** For each partition value that should raise an exception during testing, create an [ERROR ] field and add it to the test design.
- b) **Message keyword and error message list block.** For each ERROR field, make sure there is a corresponding error message in a message list block.

See the example for TRI STAGE 4 for an illustration. When the error indicators are added to the 3 partition values as indicated below, 651 test cases result :

Category	Partition Value	Fields
array size	0	[ERROR ] . . .
array size	greater than Upper Bound	[ERROR ] . . .
sort order	unspecified	[ERROR ] . . .

#### 4.2.3 TRI STAGE 3: Condition Constrained Representation

The third stage of test design creation using TRI is probably the most difficult and time consuming. Adding in the conditional statements to make sure that only the technically feasible combinations of partition values get produced in the resulting test cases often takes many iterations. Investigating exactly which combinations are valid when used together and what the expected outputs of the system should be can expose many inconsistencies and undocumented requirements .

This stage, similar to the previous one, corresponds to the fourth step in the category-partition method. The purpose of this stage is to determine the pre and post condition pairs that describe the behavior of the system under test .

To modify the existing TRI file so that the conditions are expressed, the SET and IF field keywords must be added. There will be some occasions where the addition of "don't care" partition values, or even the addition of repeat partition values with different conditions] fields attached, will be necessary in order to produce the optimum set of resulting test cases.

The following keywords and information should be added to the TRI file:

- a) **SET field keywords and post-conditions.** For each partition value that should cause a post-condition to exist if it is used in a test case create a post-condition value and append it to the inside of the [SET ] field. Use a logical AND character, "&", to separate multiple post-conditions.
- b) **IF field keywords and pre-conditions.** For each partition value that is valid only when combined with a particular partition value. in another category, append the condition value to the inside of the [IF' ] field. Use a logical AND character, "&", to separate multiple pre-conditions; a logical NOT character, "!", in front of a condition expresses that a condition should NOT exist in order for the particular partition value to be used in a resulting test case.

Again, the reader should be referred to the stage 4 discussion in section 4.2.4 for an example that has pre and post conditions. Before the LIMIT fields are added to the TRI file in stage 4, the TRI results file contains 32 test cases, which together, represent the complete functionality of the requirement being tested in this functional unit. The purpose of the fourth stage is to reduce the number of test cases even further so that testing of this functional unit takes less resources.

#### 4.2.4 TRI STAGE 4: Limit Constraint Representation

This final stage of TRI file development produces the Limit Constrained Representation of the test design. As explained in the description of the LIMIT field keyword, section 4.1.3.3, the purpose of the LIMIT field is to specify how many times a partition value can be used in the resulting set of test cases. Setting



```

* 5 partitions, 1 don't care
*
all 0's                [IF size ok]
                        [SET all same, dont care]
all the same but not 0  [IF size ok]
                        [SET all same, dont care]
all negative            [IF size ok] [SET not identical]
                        [LIMIT 4]
all positive            [IF size ok] [SET not identical]
                        [LIMIT 4]
mixed +/-0              [IF size ok] [SET not identical]
don't care              [IF !size ok]

NAME      value arrangement
SAMPLES
*
*      2 partitions, 1 don't care
*
minimum before max      [IF size ok, not identical]
maximum before min      [IF size ok, not identical]
don't care              [IF !not identical]
*

TYPE      input-Parameter: Sort Order

NAME      sort order
SAMPLES
*
*      3 partitions, 1 don't care
*
ascending                [IF size ok, not identical] [SET ascend]
descending                [IF size ok, not identical] [SET descend]
unspecified              [ERROR 3 [IF size ok]
                        [SET error, dont care]
don't care                [IF dont care]
*

MESSAGE
Sort order is not specified
*

TYPE      Output to program or change in state

NAME      result
SAMPLES
*
*      4 partition values
*
error notification       [IF error]
array unchanged           [IF dont care, not identical]
array in ascending order  [IF ascend, not identical]
array in descending order [IF descend, not identical]

```

**Figure 4-3 Test Case Results of Stage 4 Example of TRL Test Design**

Description:

```

Test Representation for SORT requirement .
File Name:      SORT.TRL
Version:        1.5 Errors/Messages/Conditions/Limits/Labels
Last Modified:  9/4/91
Modified By:    J. Hops

```

```

*****h*****.* **
Case #         1
Label:         1 .6.3.4.1

```

PARAMETERS :

```

Type:  Input-Categories for Parameter: Array

Category Name: array size
Partition Value:      0 (array unspecified)
Partition Label:      error condition
Iteration number:     1

Category Name: array values

```

```

        Partition Value:      don't care
        Partition Label:      instance value needed to pass error

    Category Name: value arrangement
        Partition Value:      don't care
        Partition Label:      instance value needed to pass error

Type: Input-Parameter: Sort Order

    Category Name: sort order
        Partition Value:      don't care
        Partition Label:      instance value needed to pass error

Type: Output to program or change in state

    Category Name: result
        Partition Value:      error notification
        Partition Label:      instance value needed to pass error

Error III: Array size of 0 is invalid or array size is unspecified.

```

\*\*\*\*\*

```

Case #      ?
Label:      ? 6.3.4.2

```

PARAMETERS :

```

Type: Input-Categories for Parameter : Array

    Category Name: array size
        Partition Value:      1 (degenerate array)
        Partition Label:      degenerate array

    Category Name: array values
        Partition Value:      don't care
        Partition Label:      valid

    Category Name: value arrangement
        Partition Value:      don't care
        Partition Label:      valid

Type: Input-Parameter: Sort Order

    Category Name: sort order
        Partition Value:      don't care
        Partition Label:      valid

Type: Output to program or change in state

    Category Name: result
        Partition Value:      array unchanged
        Partition Label:      valid

```

No error conditions exist.

\*\*\*\*\*

## 5. Conclusion

The purpose of the test design phase is to determine a set of technically feasible anti resource frugal test cases that meet the test objectives of the test plans and that verify the functional requirements of the system under test. The category-partition method can be used to determine test designs that meet this goal.

The Test Representation Language (TRL) and the TRL computer tool, used to process files written in the language, have proven very useful and efficient in implementing the category-partition method. In one project in particular at our organization, the test cases that result from the TRL tool are being used to verify the system requirements in the engineering testing stage. Detailed test procedures will be developed based on the output of the tool.

As of yet, no objective data have been collected that can be used to compare the results of the testing process changes introduced by the use of the TRL tool. However, the qualitative feedback received from both test engineers and software

designers is that the category-partition method and the TRI tool help them engineer tests rather than just perform tests. The effects of the method and the tool may be hard to quantify on an ongoing project. A way could be found to determine these effects if a small, controlled case study were to be initiated where 2 groups perform the same job - one using TRI and the category-partition method, and the other using neither.

Work is continuing in enhancing the TRI tool to meet the needs of the test engineers using it. Some keywords are being added to allow some very fine tuned control over which test cases get included in the results. We are also looking into an enhancement of the output capabilities.

In summary, the purpose and requirements of the test design phase of the testing lifecycle have been explored and defined. The category-partition method and the TRI tool are efficient ways to produce the test designs and resulting test cases needed as input into the following phase of the testing lifecycle. The Test Representation language and the TRI tool can be of use to the test engineer or programmer no matter what level of testing is being performed. More effort should be placed in gathering the necessary metrics to be able to quantify the benefits received from implementing this process. If qualitative results are enough, however, most organizations could profit from an implementation similar to the TRI tool and the category-partition method for bridging the gap between test requirements and test specifications.

#### ACKNOWLEDGEMENTS

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

#### REFERENCES

1. Balcer, M.J., Hasling, W.M., and Ostrand, T.J., "Automatic generation of test scripts from formal test specifications", Communications of the ACM, June 1989.
2. JPL Software Management Standards Package, Version 3.0, JPL D-4000, December 1988, JPL internal Document.
3. Military Standard Defense System Software Development, DOD-STD-2167A, February 1988.
4. Myers, G.J., The Art of Software Testing, Wiley Series in Business and Data Processing, John Wiley and Sons, 1979.
5. Ostrand, T.J. and Balcer, M.J., "The category-partition method for specifying and generating functional tests", Communications of the ACM, Volume 31 Number 6, June 1988.
6. Richardson, D.J., O'Malley, O., and Tittle, C., "Approaches to specification-based testing", Communications of the ACM, June 1989.
7. Weyuker, E.J., and Ostrand, T.J., "Theories of program testing and the application of revealing subdomains", IEEE Transactions on Software Engineering, Vol. SE-6, No.3, May 1980.